A reactive extension of C

Frédéric Boussinot

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1027

Programme 1

A REACTIVE EXTENSION OF C

Frédéric BOUSSINOT

Mai 1989

RR-1027

A reactive extension of C
Une extension réactive de C

Frédéric Boussinot
Ecole Nationale Supérieure des Mines de Paris
Centre de Mathématiques Appliquées
Sophia-Antipolis 06565 Valbonne France

Abstract A reactive system is a system whose behavior is to react to sequences of activations coming from the external world. Systems that can be coded as automata are examples of reactive systems. One describes an extension of the C programming language called RC (for "reactive C"). In this extension, one can easily code reactive systems. New RC notions are described, then some RC programming examples are given. In particular, one gives the complete RC code for a reflex game. The main RC notions come directly from the Esterel synchronous programming language. Finally, one compares the Esterel and RC languages.

Résumé Un système réactif est un système dont le comportement consiste en une suite de réactions à des activations périodiques provenant du monde extérieur. Les systèmes qui peuvent être codés sous forme d'automates sont des exemples de systèmes réactifs. On décrit une extension du langage C appelée RC (pour "reactive C") qui permet une programmation naturelle des systèmes réactifs. On commence par décrire les nouvelles notions introduite en RC. Plusieurs exemples de programmation sont donnés, en particulier un programme décrivant un jeu de réflexe. Les principales constructions de RC sont issues du langage Esterel qui est un langage de programmation synchrone des systèmes réactifs. Finalement, on compare les deux langages Esterel et RC.

1 Introduction

The behavior of a reactive system is to react to sequences of activations coming form the external world. Reactive systems are slaves of the outside: they do nothing unless they are activated. Finite states automata are suited to code reactive systems: automata transitions represent system activations. In fact every system that can be coded as an automaton falls into the category of reactive systems.

Esterel [3] is a programming language especially designed to program reactive systems. Esterel is based on a synchrony hypothesis: activations and reactions of a system are synchronous. Moreover, there exists a parallel operator in Esterel. Esterel programs are compiled: the compiler constructs an automaton whose behavior is equivalent to the source program.

This paper describes a new extension of the C programming language [4] for reactive systems programming. This extension is called RC for "reactive C". The new notions are reactive procedures, reactive statements, and signals. They come directly from Esterel. The intention is mainly to capture the reactive part of Esterel and to fit it to a very widespread and simple programming language.

A reactive procedure is a procedure that has an associated set of control points. The execution of such a procedure starts from these control points to reach another such set. So, a reactive procedure does not necessary react in the same way each time it is called, even in the same environment (same parameters and global variables). The behavior of a reactive procedure is built from reactive statements. An example of reactive statement is the `watching` statement which allows one to use "watchdogs" in RC. One has also introduced in RC a limited form of Esterel signals. As Esterel, RC is a deterministic language.

Presently, RC is implemented as a pre-processor which generates pure C. An idea of the translation will be given for each new notion. A small reflex game is completely coded as an example of RC programming.

2 Language description

2.1 Reactive procedures

At each call, the following C procedure prints "hello, world".

```
hello(){
        printf("hello, world\n");
}
```

Now, suppose one wants the procedure to print "hello, world" during the first call and "I repeat: hello,world" during the second call. One writes in RC:

```
rproc Hello(){
        printf("hello, world\n");
        stop;
        printf("I repeat: hello, world\n");
}
```

The `rproc` keyword introduces a *reactive procedure* definition. The behavior of a reactive procedure depends on the previous calls. A `stop` statement indicates that execution is stopped for the current call. At the next call, the execution will start from the statement following `stop`. The `stop` statement is the first example of *reactive statement*.

To call a reactive procedure, one uses the `exec` statement:

```
exec P(exp1,...,expn);
```

Of course, the argument list can be empty. In the sequel, reactive procedure calls will often be called *instants*. So, the reactive procedure `Hello` reacts at the first instant by printing "hello, world". At the second instant, it prints "I repeat: hello, world". At the next instants, the procedure is terminated and nothing is printed.

Another example of reactive statement is the `nothing` statement which does nothing and terminates. One says that `nothing` *terminates instantaneously*, that is at the first instant.

Reactive procedures are the way to write sequences of reactive statements. To execute `Bye` after `Hello` termination, one simply writes:

```
rproc Seq(){
        exec Hello();
        exec Bye();
}
```

The Bye reactive procedure is for instance:

```
rproc Bye(){
        stop;
        printf("Bye!\n");
}
```

The procedure Seq prints "hello, world" during the first instant. The control point becomes the stop in Hello. During the second instant, Seq prints "I repeat: hello, world" and then Hello terminates. In the same instant, the procedure Bye is executed and the new control point becomes the stop statement in Bye. At the third instant, Seq prints "Bye!" and terminates.

Translation in pure C. In order to illustrate the notion of reactive procedure, one gives in this paragraph a possible translation in pure C of the Hello and Seq procedures described above.

First, one has to code the control point from where the execution starts. Second, a reactive procedure must return a termination status to indicate if it is terminated.

Code for Hello and Seq would be in C:

```
#define STOP 0
#define TERMINATED 1

Hello(){
        static lab = 0;
        switch(lab){
                case 0: goto lab0;
                case 1: goto lab1;
                case 2: return TERMINATED;
        }
        lab0: printf("hello, world\n");
              lab = 1; return STOP;
        lab1: printf("I repeat: hello, world\n");
              lab = 2; return TERMINATED;
}


int Seq(){
        static lab = 0;
        int res;
        switch(lab){
                case 0: goto lab0;
                case 1: goto lab1;
                case 2: return TERMINATED;
        }

        lab0: res = Hello();
        if (res==STOP) return res;
        lab = 1;
        lab1: res = Bye();
        if (res==STOP) return res;
        lab = 2;
}
```

Notice the static variable declarations of lab for coding control points.

2.2 The Loop and Repeat statements

Never ending loops are coded with the `loop` construct. Consider the example:

```
rproc HelloLoop(){
        loop exec Hello();
}
```

The procedure `HelloLoop` prints "hello, world" at the first instant. The control point becomes the `stop` statement in `Hello`. At the second instant, `HelloLoop` starts from the previous control point and prints "I repeat: hello, world". The execution of `Hello` is then terminated; it is immediately re-started and "hello, world" is printed again. Therefore, except at the first instant, `HelloLoop` prints at each instant "I repeat: hello, world" then "hello, world", in this order.

Another example of `loop` use is the `halt` procedure which never terminates:

```
rproc halt(){
        loop stop;
}
```

Exits from loops are done using the `raise` statement (see below).

The `repeat` statement gives the way to program finite loops. For instance, the `Work` procedure is executed ten times in the following statement:

```
repeat (10) exec Work();
```

Warning! The execution of `loop <react>` is looping forever if `<react>` terminates instantaneously. For instance, consider:

```
loop nothing;
```

This `loop` statement is equivalent to:

```
l: goto l;
```

2.3 The Watching statement

It is possible to "kill" reactive statements using the `watching` instruction. Its general form is:

```
watching (E) <react1>
timeout <react2>
```

This statement behaves like `<react1>` as long as `E` is not satisfied. It terminates if `<react1>` does, while `E` is not satisfied. From the moment `E` is satisfied, the execution of `<react1>` is abandoned and the execution of `<react2>` begins. For instance, if `E` is satisfied at the first instant, `<react1>` is not executed and the `watching` behaves as `<react2>`. Therefore, the two following statements are equivalent to `<react1>`

```
#define FALSE 0
#define TRUE 1

watching (FALSE) <react1> timeout <react2>

watching (TRUE) <react2> timeout <react1>
```

As an example of `watching` use, consider the statement:

```
watching (END) exec Hello();
timeout exec Bye();
```

At the first instant there are two cases:

- The `END` condition is satisfied. The statement behaves as `Bye` and nothing is printed. The new control point becomes the `stop` statement of `Bye`.

- The END condition is not satisfied. The execution of Hello begins and "hello, world" is printed. The new control point becomes the stop of Hello.

At the second instant there are the following cases:

- The previous control point was in Bye. "Bye!" is printed and the watching statement terminates.
- The previous control point was in Hello and END is again not satisfied. "I repeat: hello, world" is printed and the watching statement terminates.
- The previous control point was in Hello and END is now satisfied. Hello is aborted and execution of Bye begins. The new control point becomes the stop in Bye. At the third instant, "Bye!" will be printed and the watching statement will terminates.

2.4 The Await statement

One waits for a condition to be satisfied using the await reactive statement. The await (C) statement terminates when C is satisfied. So the following statement is equivalent to nothing:

```
await (TRUE);
```

In the same way, the following statement is equivalent to loop stop;

```
await (FALSE);
```

One can express await with watching: the statement await (C) is equivalent to

```
watching (C) loop stop;
```

2.5 Exceptions

The execution is stopped on a raise statement, and an exception is then raised. The control can be recovered using a catch statement. For instance in the following statement, the error exception is raised if the variable X is equal to zero before the variable Y is.

```
watching (X==0)
      watching (Y==0) loop stop;
      timeout nothing;
timeout raise error;
```

To execute <react2> for recovering on an exception EXCP in <react1>, one writes:

```
catch EXCP <react1>
handle <react2>
```

For instance:

```
catch ERROR exec Work();
handle exec ReportError();
```

The repeat construct can be expressed with catch and raise statements:

```
repeat (exp) <react>
```

is equivalent to:

```
count = exp;
catch END
   loop
     { exec Test();
      <react> }
handle nothing;
```

with Test defined as:

```
rproc Test(){
   i(0==count--) raise END;
}
```

2.6 The Par statement

The par reactive statement introduces a limited form of parallelism: it allows one to execute two reactive statements during the same instant. However, the order of execution of the two instructions is fixed and specified by the syntax of the statement. The par reactive statement has the form:

```
par <react1>
    <react2>
```

There are two control points associated to each par statement. At each instant, one starts by executing <react1> then <react2>. The parallel terminates when its two components do so. If one of its components terminates, the par statement behaves like the other. At the next instant, the execution restarts from the two control points (possibly only one, if a component has terminated) where the execution was stopped.

Consider the statement:

```
par exec Hello();
    exec Bye();
```

At the first instant, it prints "hello, world". At the second instant it prints first "I repeat: hello, world" and then "Bye!". The reactive procedures hello and Bye are both terminated, so the par statement is also terminated.

To illustrate the par semantics, here are some examples:

- The two statements

```
par loop stop;
    <react>
```

  and

```
par <react>
    loop stop;
```

  are both equivalent to the sequence <react> loop stop;.

- The following statement is equivalent to <react2>:

```
catch END
   par raise END;
       <react1>
handle <react2>
```

- If <react1> terminates instantaneously, then

```
par <react1>
    <react2>
```

  is equivalent to the sequence <react1><react2>.

Warning! It may be difficult to control interferences of one component of a par statement with the other component. For example, consider the following statement:

```
par exec Hello();
    exec Hello();
```

The two messages "hello, world" and "I repeat: hello, world" are both printed at the first instant and the statement terminates instantaneously. There is only one instance of the Hello procedure which is called twice at the first instant.

2.7 The Every statement

In addition to the `loop` statement, one can write never ending loops using the `every` reactive statement. The syntax is:

```
every (C) <react>
```

Execution of `<react>` begins at the first instant `C` is satisfied and it is restarted every time `C` is satisfied. As for `loop` and `repeat` statements, one exits `every` loops using the `raise` statement.

Consider the following procedure:

```
rproc HelloEvery(){
        every (C) exec Hello();
}
```

The execution of `Hello` begins at the first instant `C` is satisfied and "hello, world" is printed.

- If `C` is no more satisfied at the next instant, "I repeat: hello, world" is printed. The procedure `Hello` will be restarted at the first instant `C` is satisfied.
- If `C` is again satisfied at the next instant, `Hello` is restarted and "hello, world" is printed one more time.

For instance, if `C` is true only at the second, fifth and sixth instants, "hello, world" is printed at the second, fifth and sixth instants; "I repeat: hello, world" is printed at the third and seventh instants; nothing is printed at the first and fourth instants, and after the seventh instant.

One can express `every` statements with `loop` and `par` statements:

```
every (C) <react>
```

is equivalent to:

```
loop {
        await (C);
        catch END
           par
              { stop; await (C); raise END; }
              { <react> loop stop; }
        handle nothing;
    }
```

Warning! The `stop` statement before the second `await(C)` statement is absolutely necessary. Without it, the overall statement would loop forever at the first instant `C` is satisfied.

Translation in pure C. One needs to reset reactive procedures to be able to restart them while they are not terminated. So, a possible pure C code for `HelloEvery` would be:

```
#define RESET 1
#define REACT 0

HelloEvery(reset)
int reset;
{
        if (reset){ Hello(RESET); return 0; }
        if (C) Hello(RESET);
        return Hello(REACT);
}
```

And `Hello` becomes:

```
Hello(reset)
int reset;
{
        static lab = 0;
        if (reset){ lab = 0; return 0; }
        switch(lab)
        .........
}
```

2.8 The Select statement

The **select** statement selects at each instant a statement to be executed. The syntax is:

```
select (C) <react1>
           <react2>
```

At each instant, **<react1>** is executed if **C** is satisfied, otherwise **<react2>** is executed. The overall statement terminates when the selected statement does so. If it does not, the execution at the next instant restarts from where it was stopped previously in the selected statement. For instance, two statements **<react1>** and **<react2>** are executed alternately, by:

```
int x = FALSE;
select (x = !x) <react1>
                <react2>
```

The **watching** statement with empty timeout can be expressed with **select**:

```
watching (C) <react>
timeout nothing;
```

is equivalent to

```
select (C) nothing;
           <react>
```

To illustrate the **select** statement, here are some examples:

- To control the execution of **<react>**by a condition **C** , one writes:

  ```
  select (C) <react>
             loop stop;
  ```

  So **<react>** is executed only when **C** is satisfied.

- Suppose now one wants to suspend execution of **<react>** when the condition SUSP is satisfied, and to restart it when the condition RST is. One writes:

  ```
  select (Control(SUSP,RST)) <react>
                             loop stop;
  ```

  with **Control** defined by:

  ```
  int Control(S,R)
  int S,R;
  {
          static act = TRUE;
          if (act==TRUE && S) return act = FALSE;
          if (act==FALSE && R) return act = TRUE;
          return act;
  }
  ```

- Using the previous `Control` function, one can program a switch between two statements `<react1>` and `<react2>`. Consider:

```
select (Control(C,C)) <react1> <react2>;
```

  The execution is switched from one statement to the other every time the `C` condition is satisfied. (In particular, if `C` is always satisfied, `<react1>` and `<react2>` are executed alternately.)

2.9 Calls of C procedures

Sometimes one wants to consider a C procedure call as a reactive statement (which of course, terminates instantaneously), for instance to trace execution. The `call` statement allows one to do so. For example:

```
loop {
   exec P();
   call printf("End of P\n");
}
```

The message "End of P" will be printed each time the body of the loop terminates.

2.10 Grammar of reactive statements

We now give the grammar for reactive statements and procedures. Reactive procedure declarations are similar to pure C procedure declarations except that reactive statements are allowed where pure C statements are. In the grammar:

- `<react>` denotes reactive statements
- `<rproc_decl>` denotes reactive procedure declarations
- `<rproc_body>` denotes reactive procedure bodies
- `<arg_decl_list>` denotes lists of arguments declarations.

```
<react> : nothing ;
        | stop ;
        | exec <name> (<expression_list>) ;
        | await (<expression>) ;
        | loop <react>
        | repeat (<expression>) <react>
        | every (<expression>) <react>
        | watching (<expression>) <react> timeout <react>
        | catch <name> <react> handle <react>
        | raise <name>;
        | par <react> <react>
        | select (<expression>) <react> <react>
        | call <name> (<expression_list>) ;
        | { <react> ... <react> }

<rproc_decl> :
        rproc <name> (<name_list>) arg_decl_list { <rproc_body> }
```

2.11 Signals

Signals gives a way to define global informations with restricted access. A signal can be emitted, read and reset. There is the (dynamically checked) restriction that once a signal has been read, no emission of it is allowed unless it is reset. So, all emissions must be done before the signal is read.

Signals are related to the instant notion: if they are reset at the end of each instant, all readers necessarily read the same informations during an instant.

More precisely, a signal is an object on which the following operations are defined:

- Emission: `emit(S);`
- Cancel of previous emissions: `reset(S);`
- Presence test: `present(S)` is an expression whose value is 1 if signal `S` has been emitted; otherwise its value is 0.
- Valued emission: `emitval(S,exp);` Signal `S` is emitted and its value becomes the value of `exp`. Signal values are always of integer type `int`.
- Use of value: `valof(S)` returns the value of `S`.

Signal declarations are of the form:

```
signal DISPLAY;
```

To assign `INPUT` value to `DISPLAY`, one writes:

```
emitval(DISPLAY,valof(INPUT));
```

Restriction of use. The major restriction on signal use is: **All emissions must be done before the first presence test of a signal and before the first use of its value.** A run-time error is raised when the restriction is violated. This is the case for the following statements:

```
emit(S); if (present(S)) emit(S);

await (present(S)); emit(S);

emitval(DISPLAY,valof(DISPLAY)+1);
```

Also, there is an error in the two following statements if `<react>` emits `S` at the first instant:

```
watching (present(S)) <react>
timeout <react1>

every (present(S)) <react>
```

2.11.1 Combine functions

A signal can be emitted with possibly different values, several times during the same instant. For instance, it is possible to write:

```
emitval(SIG,1); emitval(SIG,2);
```

By default, the last emission overrides the previous ones. So, in the preceding example, the value of `SIG` would be 2. As in Esterel, one gives the user the possibility to change this situation by associating a *combine* function to a signal. Then, emitted values will be combined using this function. One associates a combine function `f` to a signal `S` by executing the call:

```
combine(S,f);
```

Combine functions take two parameters of type `int` and return values of this type. Now, if `S` has been already emitted and has the value `v`, then after `emitval(S,e)`, `S` value becomes `f(v,e)`. As example, if `plus` denotes addition, the value of `SIG` is 3 after executing:

```
combine(SIG,plus);
emitval(SIG,1);
emitval(SIG,2);
```

Single signals. As in Esterel, one calls *single* a signal which cannot be emitted more than once without been reset. S becomes single after:

```
combine(S,abort);
```

Here, the abort procedure aborts execution. Notice that in contrast with Esterel, single signals emitted more than once are checked at run-time.

Reverting to the initial signal state. Suppose one wants to revert to the initial situation for a signal S, that is before execution of any combine(S,f) call. In this initial situation, during an instant, the last emission of S overrides the previous ones.

One reverts to the initial situation by defining the second projection as combine function for S:

```
proj2(a,b) int a,b; { return b; }
combine(S,proj2);
```

2.11.2 Counting signal occurrences

Suppose one wants to react to several signal occurrences. This is possible in RC with an auxiliary counting variable. For example in the following statement, the error exception is raised if READY is not present in less than LIMIT MS.

```
watching (present(MS) && 0 == LIMIT--) await (present(READY))
timeout raise error;
```

(Notice that LIMIT-- is executed only when MS is present, because of the semantics of && in C.)

2.11.3 Initial instant

In Esterel the watching S statement does not take in account the presence of S at the initial instant. For instance, T is emitted in the following Esterel instruction:

```
emit S;
do
   emit T
watching S
```

One can write in RC an equivalent of the Esterel watching statement using the reset operation. For instance, in the following statement, <react1> is always executed at the first instant, no matter whether S is present.

```
reset(S);
watching (present(S)) <react1> timeout <react2>
```

2.11.4 Boolean expressions on signals

One easily expresses statements reacting on boolean expressions made of several signal presence tests. For example, in the following statement, <react> is executed each time S is absent:

```
every (!present(S)) <react>
```

In the following statement, <react> is killed if S3 is present or if S1 and S2 are simultaneously present:

```
watching((present(S1) && present(S2)) || present(S3))
   <react>
timeout nothing;
```

At the present time, boolean expressions on signals are not allowed in Esterel.

2.11.5 Signals implementation

Here are some elements of signals implementation. The type `signal` is defined as:

```
typedef struct { int kind;
                 PF fun;
                 int emitted;
                 int read;
                 int value;
               } signal;
```

where PF is the C pointer to function type. The signal presence test is defined by:

```
#define present(sig) ((sig).read = 1,(sig).emitted)
```

The simple signal emission is defined as:

```
#define emit(sig) {\
      if((sig).read) _sigabort(); (sig).emitted = 1; \
}
```

The `_sigabort` function is called every time the signal restriction of use is violated. It can be user-defined. The default definition is:

```
#ifndef SIGABORT
_sigabort(){
        fprintf(stderr,"*** rc signal violation\n");
        abort();
}
#endif
```

2.11.6 Grammar of signals

One now gives the grammar of signals. In the grammar `<signal_decl>` denotes signal declarations and `<statement>` and `<expression>` denotes respectively C statements and C expressions.

```
<signal_decl> : signal <decl_list> ;

<statement> :
        ..........

      | combine (<expression>,<name>) ;
      | emit (<expression>) ;
      | emitval (<expression>,<expression>)


<expression> :
        ..........

      | present (<expression>)
      | valof (<expression>)
```

3 A reflex game

The reflex game we are going to implement is fully described in [2].

3.1 Specification

We recall the reflex game specification: A new game is started afresh every time a coin is inserted. Each game is composed of a fixed number of measures. A measure starts when the player press the READY button. After a random delay, the GO lamp is switched on. The player must then press the END button as fast as he can. The reflex time between GO and END is displayed. When all the measures are done, the average reflex time is displayed.

There is a beep in case of player mistakes: if he presses END instead of READY to begin a measure, or if he presses READY instead of END during a measure. On the other hand, the game tilts and terminates if the player takes too much time to press READY or END when he is supposed to do so, or if he presses END before the moment GO is switched on (he is cheating!).

3.2 RC code

Here is the RC code:

```
extern MEASURE_NUMBER, PAUSE_LENGTH, LIMIT_TIME;

signal READY, COIN, END,
       DISPLAY, GO, GAME_OVER, TILT, RING_BELL;

static int TOTAL_TIME;

/* After printing a quick description, the game is started
   afresh whenever a coin is inserted. After a fixed number of
   measures, the final display is done. The game is over when
   there is an error. */
rproc GAME(){
   PrintOut("A reflex game ... c to start, q to stop.\r\n");
   PrintOut("Press e as fast as possible after GO!.\r\n");
   every (present(COIN))
      catch error
         { repeat (MEASURE_NUMBER) exec measure();
           exec finalDisplay();
         }
      handle call abnormal_game_over();
}

/* TILT and GAME_OVER are displayed in case of error. */
abnormal_game_over(){
   TOTAL_TIME = 0;
   emit(TILT);
   emit(GAME_OVER);
}

/* One prints a prompt message after one instant (for a good
   printing of scores), then phase 1 and 2 are done. During the
   first phase, one waits for READY. During the second, one waits
   for STOP. */
rproc measure(){
   stop;
   PrintOut("press r when ready\r\n");
   exec phase1(); exec phase2();
}

/* Phase1: One waits for READY and the bell rings when END is pressed.
   There is an error if the player takes too much time to press READY.
```

```
   Then, one waits for a random delay and also tests END. */
rproc phase1(){
   static COUNTER;
   COUNTER = LIMIT_TIME;
   watching (0 == COUNTER--)
        watching (present(READY)) loop exec beepOn(END);
        timeout nothing;
   timeout raise error;
   COUNTER = RANDOM();
   watching (0 == COUNTER--) { stop; loop exec testEnd(); }
   timeout nothing;
}

/* After one instant, the bell rings if a signal is present. */
rproc beepOn(s)
signal s;
{
        stop;
        if (present(s)) emit(RING_BELL);
}

/* There is an error if END is present. The bell rings when READY is
   pressed. */
rproc testEnd(){
        if(present(END)) raise error;
        exec beepOn(READY);
}

/* Phase 2: GO is turned on. Then one waits for END and the bell rings
   when READY is pressed. There is an error is the player takes too
   much time to press END. Finally, the reflex time is displayed. */
rproc phase2(){
   static COUNTER;
   COUNTER = 0;
   emit(GO);
   watching (LIMIT_TIME == COUNTER++)
           watching (present(END)) loop exec beepOn(READY);
           timeout nothing;
   timeout raise error;
   emitval(DISPLAY,COUNTER);
   TOTAL_TIME += COUNTER;
}

/* After a fixed delay, the average reflex time is printed. */
rproc finalDisplay(){
   static COUNTER;
   COUNTER = 0;
   await (COUNTER++ == PAUSE_LENGTH);
   PrintOut("**** final ");
   emitval(DISPLAY, TOTAL_TIME / MEASURE_NUMBER);
   emit(GAME_OVER);
   TOTAL_TIME = 0;
}
```

3.3 Simulation

The simulation environment is realized under Unix BSD4.2. The "c" key represents COIN, the "r" key represents READY and the "e" key represents END. There are four measures and one gives suitable values to PAUSE_LENGTH and LIMIT_TIME constants, and to values returned by RANDOM.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/time.h>

fd_set readfds;
struct timeval t;

int MEASURE_NUMBER = 4;
int PAUSE_LENGTH = 1000;
int LIMIT_TIME = 10000;

RANDOM(){
        return random()%10000;
}

extern signal READY, COIN, END, DISPLAY,
              GO, GAME_OVER, TILT, RING_BELL;

main(){
        TheBegining();
        for(;;){
                AnalyseInput();
                exec GAME();
                AnalyseOutput();
                ResetSignals();
        }
}

/* The following is necessary as select is a RC keyword */
#define C_SELECT select

AnalyseInput(){
        FD_SET(0,&readfds);
        if(C_SELECT(1,&readfds,0,0,&t)){
                switch(getchar()){
                        case 'c': emit(COIN); break;
                        case 'r': emit(READY); break;
                        case 'e': emit(END); break;
                        case 'q': TheEnd();
                }
        }
}

AnalyseOutput(){
        if(present(DISPLAY)){
                printf("score: %d",valof(DISPLAY));
                PrintOut("\r\n");
        }
        if(present(GO)) PrintOut("GO!\r\n");
        if(present(TILT)) PrintOut("\07TILT!!\r\n");
        if(present(GAME_OVER)) PrintOut(
            "Game over. Press c to restart, q to stop.\r\n");
```

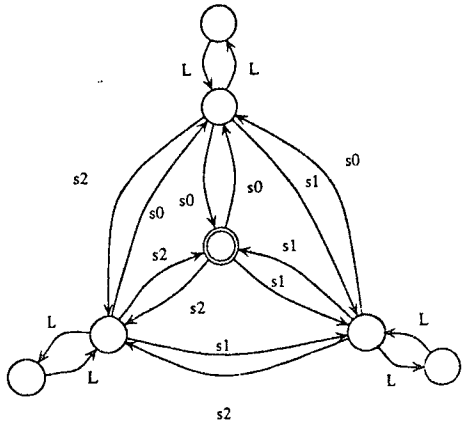
```
        if(present(RING_BELL)) PrintOut("\07");
}

TheBegining(){
        system("stty raw -echo");
        FD_ZERO(&readfds);
        t.tv_sec = t.tv_usec = 0;
}

TheEnd(){
        PrintOut("It's more fun to compete ...\r\n");
        system("stty -raw echo"); exit(0);
}

ResetSignals(){
        reset(READY);
        reset(COIN);
        reset(END);
        reset(DISPLAY);
        reset(GO);
        reset(GAME_OVER);
        reset(RING_BELL);
        reset(TILT);

}

/* For good printing */
PrintOut(ch)
char* ch;
{
     printf("%s",ch);
     fflush(stdout);
}
```

4 Automata descriptions

4.1 Specification

Consider the following specification: There are N+1 buttons $L, S_1, ..., S_N$. Button S_i is preselected when pressed. One returns to the initial state if S_i is pressed another time. When S_i is preselected, it becomes selected when the lock button L is pressed. Then, the only possible thing to do when S_i is selected is to press L another time; then S_i returns to the preselected mode.

In fact, one wants to describe an automaton of the following form (N equal 3 in the drawing):



4.2 RC code

One supposes that N holds the number of buttons. The RC code is:

```
#define NONE -1

signal L, S[N];

rproc Entry(){
        printf("nothing is preselected\r\n");
        loop exec poll();
}

rproc poll(){
        static i;
        if (NONE == (i = PresentS())) stop;
        else{
        exec preselect(i);
                if (present(S[i])){
                        printf("end of preselection of %d\r\n",i);
                        printf("nothing is preselected\r\n");
                        stop;
                }
        }
}

rproc preselect(i)
int i;
{
        printf("preselection of %d\r\n",i);
        catch SEL
            loop exec awaitSelection(i);
        handle nothing;
}

rproc awaitSelection(i)
int i;
{
        stop;
        if (present(L)) exec selected(i);
        if (NONE != PresentS()) raise SEL;
}

rproc selected(i)
int i;
{
        printf("%d selection\r\n",i);
        stop; await (present(L));
        printf("end of selection of %d\r\n",i);
}

PresentS(){
        int i;
        for(i=0;i<N;i++) if(present(S[i])) return i;
        return NONE;
}
```

4.3 C code

For comparison, here is a direct C coding. The automaton states are represented by two global variables STATE and FROM. The variable FROM has the value i when S_i is selected.

```
#define BLOQ -1
#define INITIAL -1
#define LOCKED -2
#define NONE -1

int L, S[N], FROM, STATE = INITIAL;

Entry(){
   int sig;
   for(;;){
        if(STATE==INITIAL){
                poll();
                return;
        }else if(STATE==LOCKED){
                selected();
                return;
        }else{
                if(BLOQ == (sig = preselect())) return;
        }
   }
}

poll(){
        int i;
        if (NONE != (i = PresentS())){
                printf("preselection of %d\r\n",i);
                STATE = i;
        }
}

preselect(){
        int sig;
        if (L){
            printf("selection of %d\r\n",STATE);
            FROM = STATE;
            STATE= LOCKED;
        }else if (NONE != (sig=PresentS())){
                if (sig == STATE){
                    printf("end of preselection of %d\r\n",STATE);
                    STATE = INITIAL;
                    printf("nothing is preselected\r\n");
                }else{
                    printf("preselection of %d\r\n",sig);
                    ResetSignals();
                    return STATE = sig;
                }
        }
        return BLOQ;
}

selected(){
        if(L){
```

```
                STATE = FROM;
                printf("end of selection of %d\r\n",FROM);
        }
}


PresentS(){
        int i;
        for(i=0;i<N;i++) if(S[i]) return i;
        return NONE;
}
```

5 Use of the Par statement

Consider the following specification: there are two processes in a system; each process has a proper job to do; moreover it has to monitor the other process to detect its failure. Here is the code for the first process (the code for the second process is similar):

```
signal FAIL1,FAIL2;

static OK1,OK2;

signal P1OUT, P2OUT;

rproc Sys(){
        par exec process2();
            exec process1();
}

rproc process1(){
    catch error
        par loop exec work1();
            loop exec observeProcess2();
    handle loop stop;
}

rproc work1(){
        if(present(FAIL1)) raise error;
        OK1 = 1;
        stop;
}

rproc observeProcess2(){
        static COUNT;
        COUNT = PL;
        watching (0==COUNT--) await(OK2);
        timeout exec process2OUT();
        OK2 = 0;
        stop;
}

rproc process2OUT(){
        emit(P2OUT);
        loop stop;
}
```

6 Comparison with Esterel

Esterel [3] is a synchronous language for programming reactive systems. In Esterel there is a powerful parallel operator. To compile an Esterel program means to construct an automaton equivalent to the program. Several tools use the generated automata to produce code in others programming languages or to produce entries for proofs or validation systems.

The reactive statement notion described here comes directly from Esterel. However, there are several syntactic differences. Some of them are:

- Conditions in Esterel reactive statements are restricted to signal presence tests.
- Signal tests are Esterel statements but RC expressions.
- The `halt` statement is a kernel statement in Esterel, and it is derived in RC (notice that `stop` or `select` statements cannot be directly expressed in Esterel).

The main differences between Esterel and RC concern parallelism and execution.

Parallelism. In contrast with Esterel, there is no real parallelism operator in RC. In particular, the par statement is not commutative.

Consider the following Esterel program:

```
  present S1 then emit S2 end
||
  emit S1; present S2 then emit S3 end
```

First S1 must be emitted, then it is tested and S2 is emitted; finally S2 is tested and S3 is emitted. This program illustrate a possibility of communication, called "instantaneous dialogue". In the same instant there is a communication from the parallel first branch to the second one, and conversely, from the second to the first. The Esterel compiler has to find a proper interleaving allowing the dialogue to take place.

Instantaneous dialogues do not exist in RC where interleaving of instructions in the same instant is not possible.

Furthermore in Esterel, signals are the only way to synchronize branches of parallel instructions: no global variables are allowed. In RC, communication is not so structured and global variables and side-effects are allowed, as in C.

Moreover, causality cycles (see [1]) are simplified in RC: there is no equivalent to cycles coming from parallelism as in the following Esterel program:

```
  present S1 else emit S2 end
||
  present S2 else emit S1 end
```

The only cycles are those where a signal is emitted after it has been read (i.e. its presence is tested or its value is used). Contrary to Esterel, these cycles are not statically detected; they produce run-time errors. (In the same way, "single" signals emitted more than once are dynamically checked.)

Execution. RC programs are directly executed in contrast with Esterel where one executes finite automata generated by the compiler. At the present time, there is no possibility in RC to produce entries for proof or validation tools as for example, the AUTO [6] system.

7 Implementation

There exists an experimental RC implementation as a C pre-processor. This implementation is written in C++ [5] using yacc and lex. The reflex game is compiled by:

```
rc game.rc > game.c
rc env.rc > env.c
cc game.c env.c -o game
```

To illustrate the pre-processor job, consider the following statement:

```
watching (cond) exec normal();
timeout exec abnormal();
```

The pre-processor generates approximately the following C text:

```
{
   int _r;
   if(0 == _v3 &&  (cond)){
       normal(_RESET);
       _v3 = 1;
   }
   if (_v3){
       _v2 = abnormal(_REACT);
       _r = _v2;
        if (_r == _TERMINATED){
               _v3 = 0;
               abnormal(_RESET);
        }
   }else{
         _v1 = normal(_REACT);
         _r = _v1;
         if (_r ==_TERMINATED){
                _v3 = 0;
                normal(_RESET);
         }
         _v4 = _r;
  }
  if(_v4 != _TERMINATED) return _v4;
  else{
         normal(_RESET);
         abnormal(_RESET);
         _v3 =0;
  }
}
```

8 Conclusion

This paper describes a new C extension for programming reactive systems. The use of reactive statements is natural when program behaviors are defined by reference to an instant notion. Several such programs are given in the paper. It would be interesting to generate automata from RC programs, coding the reactive parts of the programs. Transitions of such automata would be pure C statements. It seems also interesting to extend C++ in the same way as it has been done for the pure C language.

Acknowledgements I would like to thank G. Berry for its remarks on the previous versions of this paper.

Contents

References

[1] Gérard Berry, Georges Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, INRIA Report 842, 1988. To appear in Science of Computer Programming.

[2] *ESTEREL v2.2 Programming Examples*, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1987.

[3] *ESTEREL v3 Manuals*, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1988.

[4] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Prentice–Hall Software Series.

[5] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company.

[6] Didier Vergamini, *Vérification de réseaux d'automates finis par équivalences observationnelles: le système AUTO*, Thèse de doctorat, Université de Nice, 1987.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique